

Contenido

BOTONES	65
Raised Buttons	66
Manejar eventos botón.....	70
Flat Buttons	72
Floating Action Button	74
IMÁGENES, ETIQUETAS Y CUADROS DE TEXTO.....	77
ImageView.....	77
TextView.....	77
EditText	77
Etiquetas Flotantes (Floating Labels)	85
Etiquetas predictivas: (AutoCompleTextView)	90
CHECKBOX.	92
RADIOBUTTON	95
TOAST	97
SNACKBAR	98
Snackbar sencillo	99
Snackbar con acción	99
Descartar SnackBar	100

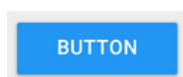
4.

Interfaz de Usuario I

BOTONES

Los botones forman una parte funcional muy importante de cualquier aplicación. Existen tres tipos estándar de botones:

- Floating Action Button, botón circular con una acción muy concreta en nuestra aplicación.
- Raised Button: botón con relieve con efecto de pulsación.
- Flat Button: botón sin relieve ni efecto de pulsación

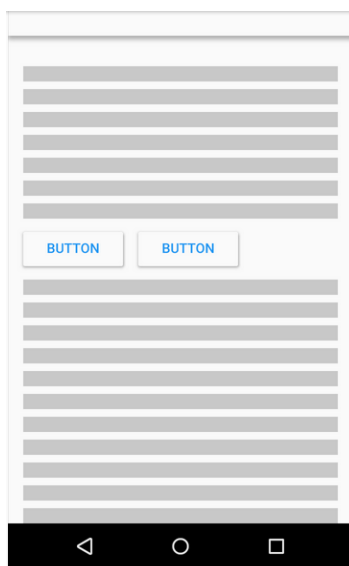


La elección de un botón u otro va a depender de la importancia que se le quiera dar al botón, del número de elementos contenedores de la pantalla y del tipo de layout.

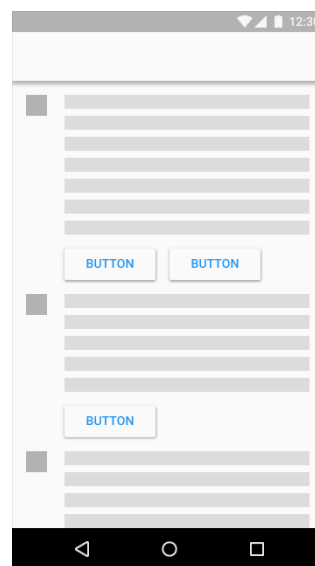
Se recomienda utilizar los FlatButton en los cuadro de diálogos, en botones en línea al finalizar la pantalla o en botones que son persistentes o que estarán siempre disponibles en nuestra aplicación.



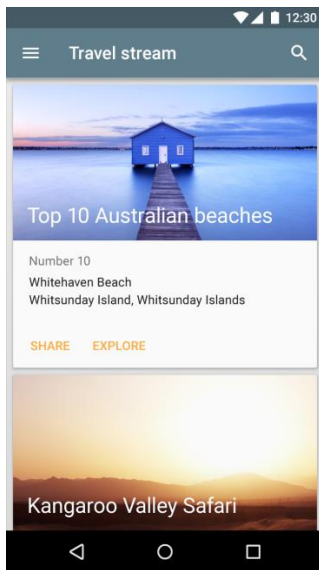
OK



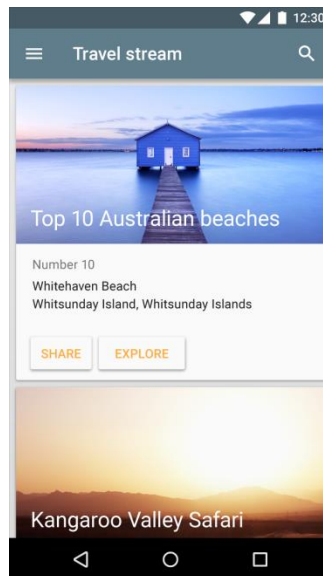
OK



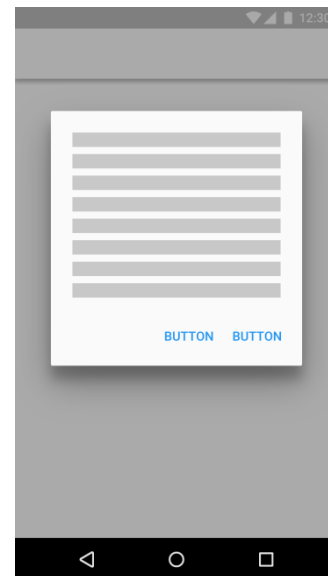
OK



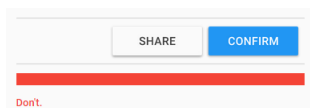
OK



No hacer



OK



No hacer en caso de botones persistentes

Raised Buttons

Un botón es un control con texto o imagen que realiza una acción cuando el usuario lo presiona. La clase Java que lo represente es [Button](#) y puedes referirte a él dentro de un layout con la etiqueta `<Button>`.

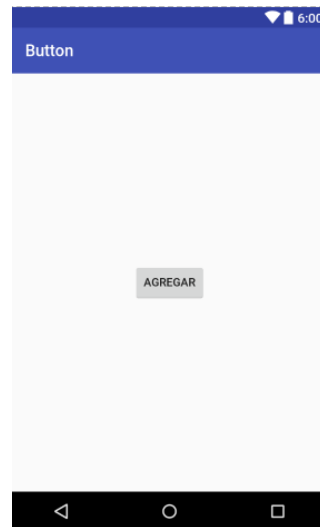
Crear un nuevo proyecto llamado Button. Editamos el archivo `activity_main`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="com.example.profesor.button.MainActivity">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="Agregar" />

</RelativeLayout>
```

El texto que quiero que aparezca dentro del botón lo especifico en el atributo *Android:text*. Este código debería mostrar en la ventana **Preview** la siguiente imagen:



Algunos atributos que permiten modificar esta View son:

Atributo	Descripción
<code>android:text</code>	Permite cambiar el texto de un botón
<code>android:background</code>	Se usa para cambiar el fondo del botón. Puedes usar un recurso del archivo <code>colors.xml</code> o un <i>drawable</i> .
<code>android:enabled</code>	Determinar si el botón está habilitado ante los eventos del usuario. Usa <code>true</code> (valor por defecto) para habilitarlo y <code>false</code> en caso contrario.
<code>android:gravity</code>	Asigna una posición al texto con respecto a los ejes x o y dependiendo de la orientación deseada. <i>Por ejemplo:</i> Si usas <code>top</code> , el texto se alinearé hacia el borde superior.
<code>android:id</code>	Representa al identificador del botón para diferenciar su existencia de otros views.
<code>android:onClick</code>	Almacena la referencia de un método que se ejecutará al momento de presionar el botón.
<code>android:textColor</code>	Determina el color del texto en el botón
<code>android:drawable*</code>	Determina un drawable que será dibujado en la orientación establecida. <i>Por ejemplo:</i> Si usas el atributo <code>android:drawableBottom</code> , el drawable será dibujado debajo del texto.

Por defecto el texto del botón estará en mayúsculas, pero si quieres deshabilitar esta característica usa el valor `false` en el atributo `android:textAllCaps`.

Lo habitual es asignar los textos a las Views a través del archivo `res/values/strings`:

```
<resources>
    <string name="app_name">Button</string>
    <string name="button">Agregar</string>
</resources>
```

Y asignar el atributo `text` del botón como sigue:

```
android:text="@string/button"
```

Es posible cambiar el color de fondo del botón y asignarle el color primario de nuestra aplicación con el atributo:

```
android:background="@color/colorPrimary"
```

Consiguiendo el siguiente efecto visual:



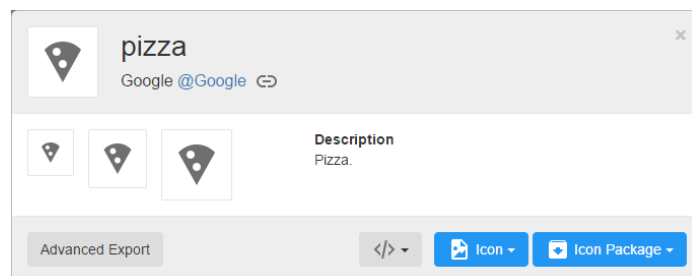
El problema es que esta definición hace que desaparezca el efecto redondeado y elevación de MaterialDesign. Para que esto no suceda utilizamos el atributo:

```
app:backgroundTint="@color/colorPrimary"
```

Siendo el resultado:

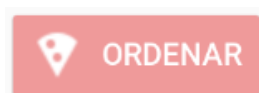


En la sección de atributos vimos que existen atributos con la forma `android:drawable*` para alinear una imagen al texto de un botón. Para comprobar en un ejemplo descargamos el siguiente [drawable de pizza](#):



Descargamos un recurso `icon` y lo guardamos en la carpeta `drawable` de nuestro proyecto.

En nuestro caso añadimos un botón debajo del ya existente y los alineamos en el centro, el color de texto lo ponemos en blanco y un fondo de color rosado, además colocamos la imagen a la izquierda del texto (`android:drawableLeft`):



El código de este botón sería:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableLeft="@drawable/ic_pizza"
    android:drawablePadding="8dp"
    android:id="@+id/botonImagen"
    android:textColor="@android:color/white"
    app:backgroundTint="#ef9a9a"
    android:text="Ordenar"
    android:layout_below="@+id/button"
    android:layout_centerHorizontal="true" />
```

Es posible también especificar un botón sin texto y solo imagen con un control de tipo *ImageButton*. Para asignar la imagen al control se utiliza el atributo *android:src*.

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_pizza"
    app:backgroundTint="#ef9a9a"
    android:drawablePadding="8dp"
    android:id="@+id/buttonImage"
    android:layout_below="@+id/botonImagen"
    android:layout_centerHorizontal="true" />
```

Para mantener el aspecto le damos también un color de fondo rosado. El resultado:



Es posible también la utilización de un control de tipo *ToggleButton* (subclase de *Button*), que permite visualizar el estado de chequeo o no de un botón.

El control se define en XML de la siguiente forma:

```
<ToggleButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/toggleButton"
    android:textOn="@string/string_on"
    android:textOff="@string/string_off"
    android:textColor="@android:color/white"
    app:backgroundTint="#ef9a9a"
    android:layout_below="@+id/buttonImage"
    android:layout_centerHorizontal="true" />
```

En el caso de un botón de tipo **ToggleButton** suele ser de utilidad conocer en qué estado ha quedado el botón tras ser pulsado, para lo que podemos utilizar su método `isChecked()`. Lo veremos en el siguiente punto: gestión de eventos de los botones.

Un control **Switch** es muy similar al *ToggleButton* anterior, donde tan sólo cambia su aspecto visual, que en vez de mostrar un estado u otro sobre el mismo espacio, se muestra en forma de deslizador o interruptor.



```

<Switch
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/switchButton"
    android:textOn="@string/string_on"
    android:text="@string/string_off"
    android:textColor="@android:color/white"
    android:background="#ef9a9a"
    android:layout_below="@+id/toggleButton"
    android:layout_centerHorizontal="true"
    android:checked="true"/>

```

Manejar eventos botón

Para gestionar la pulsación realizada sobre un botón tenemos varias posibilidades: usar el atributo *onClick* de la vista, usar un escuchador anónimo e implementar la el escuchador sobre el contenedor principal (activity, fragment,...). Veamos cada una de estas soluciones.

Atributo *onClick* de la vista

Este atributo permite asignar un método al botón que se ejecutará cuando este se pulse. Este método tiene que ser público, de tipo void, tiene que recibir un parámetro de tipo *View* y tiene que declararse en la actividad donde se define.

Vamos a definir la acción correspondiente al primer botón que definimos en el ejemplo, el botón *Agregar*.

```

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:textAllCaps="false"
    android:text="@string/button"
    app:backgroundTint="@color/colorPrimary"
    android:onClick="clickAgregar"/>

```

Ahora hay que incluir el código de dicho método en el archivo .java correspondiente a la activity principal, *MainActivity.java*.

```

package com.example.profesor.button;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void clickAgregar(View v){
        Toast.makeText(this, "Pulsaste Agregar", Toast.LENGTH_SHORT).show();
    }
}

```

Lo único que hacemos es visualizar un *toast* en la pantalla del dispositivo. Más adelante daremos más detalles sobre este elemento.

Escuchador o *listener* anónimo

Vamos a implementar un *listener* anónimo sobre el botón Ordenar del proyecto que estamos desarrollando para este punto.

Lo primero que tenemos que hacer es crear una referencia de tipo *Button* y asociarla al elemento de la vista con el que queremos trabajar, definir el *listener* asociado al botón e implementar las acciones que se deseen cuando se realice la pulsación del botón:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button botonOrdenar=(Button)findViewById(R.id.botonImagen);
    botonOrdenar.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            //acciones a realizar
            Toast.makeText(getApplicationContext(), "Pulsaste Ordenar", Toast.LENGTH_SHORT).show();
        }
    });
}
```

Implementación del escuchador o *listener* sobre la vista principal

La alternativa siguiente consiste en implementar la interfaz *OnClickListener* como parte de la vista, es decir, en el archivo .java que va a gestionar la actividad. Esta alternativa es la recomendada por *Android*, al tener menos gasto de memoria. Sin embargo, si buscáis información y código en la red, vais a ver que el *listener* anónimo es muy utilizado.

Vamos a implementar este método sobre los botones *Ordenar* e *ImageButton*.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button botonOrdenar=(Button)findViewById(R.id.botonImagen);
    ImageButton botonImagen=(ImageButton)findViewById(R.id.buttonImage);
    botonOrdenar.setOnClickListener(this);
    botonImagen.setOnClickListener(this);
}

public void onClick(View v){
    switch (v.getId()) {
        case R.id.botonImagen:
            Toast.makeText(getApplicationContext(), "Pulsaste Ordenar", Toast.LENGTH_SHORT).show();
            break;
        case R.id.buttonImage:
            Toast.makeText(getApplicationContext(), "Pulsaste Imagen", Toast.LENGTH_SHORT).show();
            break;
        default:
            break;
    }
}
```


Asignamos la escucha a los botones con `setOnClickListener(this)` y posteriormente implementamos el método de la interfaz `onClick(View v)`. Este método analiza qué `view` ha sido pulsada y ejecuta las acciones correspondientes a la misma.

Implementación del escuchador sobre *ToggleButton*

Suele ser de utilidad conocer en qué estado ha quedado el botón tras ser pulsado, para lo que podemos utilizar su método `isChecked()`.

```
btnToggle = (ToggleButton)findViewById(R.id.toggleButton);
btnToggle.setOnClickListener(new View.OnClickListener() {
    public void onClick(View arg0)
    {
        if(btnToggle.isChecked())
            Toast.makeText(getApplicationContext(), "Pulsaste Toggle a ON", Toast.LENGTH_SHORT).show();
        else
            Toast.makeText(getApplicationContext(), "Pulsaste Toggle a OFF", Toast.LENGTH_SHORT).show();
    }
});
```

Implementación del escuchador sobre *Switch*

Cuando trabajamos con este control tipo de control se puede interaccionar con él o bien haciendo un *click* o realizando un desplazamiento sobre el mismo. La forma de detectar estos eventos es a través de dos escuchadores distintos. Uno el visto anteriormente para el `onClick` del *ToggleButton* y otro a través de `onCheckedChanged`. Veamos el código de su implementación.

```
btnSwitch = (Switch)findViewById(R.id.switchButton);
btnSwitch.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if(btnSwitch.isChecked())
            Toast.makeText(getApplicationContext(), "Pulsaste Switch a ON", Toast.LENGTH_SHORT).show();
        else
            Toast.makeText(getApplicationContext(), "Pulsaste Switch a OFF", Toast.LENGTH_SHORT).show();
    }
});

btnSwitch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (isChecked) {
            Toast.makeText(getApplicationContext(), "Desplazaste Switch a ON", Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(getApplicationContext(), "Desplazaste Switch a OFF", Toast.LENGTH_SHORT).show();
        }
    }
});
```

La lógica de un programa nos dice que el código tras la pulsación de estos eventos debe ser el mismo, nosotros hemos puesto dos mensajes diferentes en el *Toast* para comprobar que la gestión es diferente según que escuchador lo gestione (aunque no sería lo razonable en ejecución real).

Flat Buttons

Otro tipo de botón *Android* es *FlatButton*. Este tipo de botones se utiliza para evitar una estratificación gratuita. Por ejemplo, se sugiere utilizar este tipo de botón en un *CardView*, en lugar de botón normal o *RaisedButton*. El objetivo principal de este botón es minimizar la distracción del usuario del contenido.

Para convertir un botón en *FlatButton* hay que añadir `style="?android:attr/borderlessButtonStyle"` en el layout donde se define el botón, dentro de la definición del botón.

```
<Button
    android:id="@+id/buttonFlat"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:textAllCaps="false"
    android:text="@string/buttonFlat"
    android:layout_below="@+id/switchButton"
    style="?android:attr/borderlessButtonStyle"/>
```

Realmente estamos aplicando un estilo para conseguir el efecto deseado sobre el botón.

Veamos la aplicación de un estilo propio en nuestro *FlatButton*, teniendo en cuenta que esta técnica se puede utilizar para cualquier otro control.

En primer lugar definimos en el archivo `styles.xml` un nuevo recurso tal y como se ve en la imagen siguiente:

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
    <style name="MyButton" parent="Theme.AppCompat.Light">
        <item name="colorControlHighlight">@color/colorAccent</item>
    </style>
</resources>
```

Y aplicamos este estilo con el atributo *theme* en la definición del botón:

```
<Button
    android:id="@+id/buttonFlat"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:textAllCaps="false"
    android:text="@string/buttonFlat"
    android:layout_below="@+id/switchButton"
    style="?android:attr/borderlessButtonStyle"
    android:theme="@style/MyButton"/>
```

Podríamos aplicar el siguiente estilo al control *Switch* y probar su funcionamiento:

```
<style name="MySwitch" parent="Theme.AppCompat.Light">
    <!-- active thumb & track color (30% transparency) -->
    <item name="colorControlActivated">@color/indigo</item>

    <!-- inactive thumb color -->
    <item name="colorSwitchThumbNormal">@color/pink</item>

    <!-- inactive track color (30% transparency) -->
    <item name="android:colorForeground">@color/grey</item>
</style>
```

Floating Action Button

Lo botones flotantes (Floating Action Button / FAB) son un nuevo “tipo de botón” que apareció a raíz de la nueva filosofía de diseño Android llamada *Material Design*. Con la librería Design Support Library, añadir este tipo de botones a nuestras aplicaciones es algo de lo más sencillo, y además aseguramos su compatibilidad no sólo con Android 5.x sino también con versiones anteriores. En esta librería se incluye un nuevo componente llamado `FloatingActionButton` con la funcionalidad y aspecto deseados.

Lo primero que tendremos que hacer para utilizarlo será añadir la librería indicada a nuestro proyecto, `android.support.design.widget.FloatingActionButton`. Para ello deberemos añadir a nuestro proyecto el módulo de dependencia correspondiente. Para ello abrir el menú *File/Project Structure* y seleccionar app y después la pestaña *Dependencies* y añadir (puede que la versión no sea la misma debido a las actualizaciones que se realizan en el sistema):

```
'com.android.support:design:28.0.0'
```

Como dice Google en la documentación del Material Design, un *float action button* es un botón para destacar una acción en tu app. Se caracteriza por tener una forma circular, un icono interno que representa la acción, una reacción de superficie y la capacidad de cambiar de forma, desplazarse e interactuar con otros elementos.

En el layout en el cual queramos añadir nuestro botón flotante (activity_main del proyectoFAB), deberemos hacer algo parecido a lo siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        />
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|right"
        android:layout_margin="@dimen/margenes_fab"
        android:src="@drawable/ic_add"
        app:fabSize="normal"
        app:backgroundTint="@color/colorPrimary"/>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Para la gestión del evento *onClick* sobre el FAB, podemos decidir de nuevo por lo métodos comentados anteriormente, en nuestro caso por un escuchador anónimo:

```

package com.example.profesor.proyectofab;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast.makeText(getApplicationContext(), "Pulsaste FAB", Toast.LENGTH_SHORT).show();
            }
        });
    }
}

```

En la gestión de los FAB existen diferentes animaciones que permiten hacer nuestra aplicación más amigable y atractiva, entre ellos podemos destacar la rotación del icono y la aplicación de una animación en escala. Veamos cómo aplicar el primero efecto a nuestro FAB tras la pulsación del botón.

Para ello disponemos de una variable *boolean* que determina el estado del botón (si ha sido clikeado o no). Comprobamos que la versión de Android es Lollipop y aplicamos la rotación de 45º o volver al estado inicial (0) según el estado de la bandera. El objeto *Interpolator* permite definir la velocidad de una animación.

```

package com.example.profesor.proyectofab;

import ...

public class MainActivity extends AppCompatActivity {
    boolean click = false;

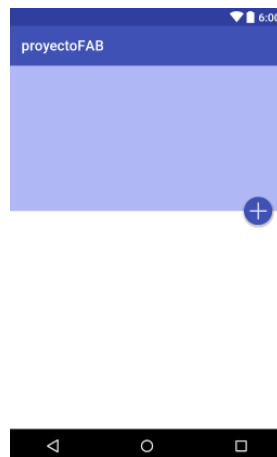
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                click = !click;
                if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
                    Interpolator interpolador = AnimationUtils.loadInterpolator(getApplicationContext(),
                        android.R.interpolator.fast_out_slow_in);

                    view.animate()
                        .rotation(click ? 45f : 0)
                        .setInterpolator(interpolador)
                        .start();
                    Toast.makeText(getApplicationContext(), "Pulsaste FAB", Toast.LENGTH_SHORT).show();
                }
            }
        });
    }
}

```

El FAB también se utiliza como elemento diferenciador entre dos *layouts*:



Para poder aplicare este efecto es necesario utilizar un *CoordinatorLayout*. Dentro de él definimos las dos vistas (*Layout*), finalmente en la definición del *FAB* con el atributo *app:layout_anchor*, fijamos a qué vista se ancla el botón. Archivo *main_activity_v2*.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:id="@+id/viewA"
            android:layout_weight="0.6"
            android:background="@color/colorPrimaryDark"
            android:orientation="horizontal"/>
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:id="@+id/viewB"
            android:layout_weight="0.4"
            android:background="@android:color/white"
            android:orientation="horizontal"/>
    </LinearLayout>
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="16dp"
        android:clickable="true"
        android:src="@drawable/ic_add"
        app:backgroundTint="@color/colorPrimary"
        app:layout_anchor="@id/viewA"
        app:layout_anchorGravity="bottom|right|end"/>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

IMÁGENES, ETIQUETAS Y CUADROS DE TEXTO

En este apartado nos vamos a centrar en otros tres componentes básicos imprescindibles en nuestras aplicaciones: las imágenes (*ImageView*), las etiquetas (*TextView*) y por último los cuadros de texto (*EditText* y *TextInputLayout*).

ImageView

El control *ImageView* permite mostrar imágenes en la aplicación. La propiedad más interesante es `android:src`, que permite indicar la imagen a mostrar. Nuevamente, lo normal será indicar como origen de la imagen el identificador de un recurso de nuestra carpeta `/res/drawable`, por ejemplo `android:src="@drawable/unaimagen"`. Además de esta propiedad, existen algunas otras útiles en algunas ocasiones como las destinadas a establecer el tamaño máximo que puede ocupar la imagen, `android:maxWidth` y `android:maxHeight`.

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imageView"
    android:src="@mipmap/ic_launcher" />
```

En la lógica de la aplicación, podríamos establecer la imagen mediante el método ***setImageResource(...)***, pasándole el ID del recurso a utilizar como contenido de la imagen.

```
ImageView imagen=(ImageView)findViewById(R.id.imageView);
imagen.setImageResource(R.mipmap.ic_launcher);
```

TextView

El control *TextView* se utiliza para mostrar un determinado texto al usuario. Al igual que en el caso de los botones, el texto del control se establece mediante la propiedad `android:text`. A parte de esta propiedad, la naturaleza del control hace que las más interesantes sean las que establecen el formato del texto mostrado, que al igual que en el caso de los botones son las siguientes: `android:background` (color de fondo), `android:textColor` (color del texto), `android:textSize` (tamaño de la fuente) y `android:typeface` (estilo del texto: negrita, cursiva, ...).

```
<TextView android:id="@+id/LblEtiqueta"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/escrIBE_algo"
    android:background="#ff1ca5ff"
    android:typeface="monospace"/>
```

EditText

El control *EditText* es el componente de edición de texto que proporciona la plataforma Android. A continuación veremos cómo cambiar los tipos de entrada de un *EditText*, modificar el texto de fondo (*hint*), tamaño de texto y color de la línea, manejar sus eventos, autocompletar, etc.

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="com.example.profesor.proyectoentradatexto.MainActivity">

    <EditText
        android:id="@+id/campo_texto"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:hint="Texto de entrada" />
</RelativeLayout>

```

En la anterior definición se centra un *EditText* en el *RelativeLayout*, cuyo ancho se ajusta al padre y el alto al contenido. Además se usa el texto auxiliar “*Texto de entrada*” en el atributo *android:hint*. El resultado sería algo así:



El atributo *android:inputType* condiciona la entrada de texto al usuario para ingresar caracteres acordes al requerimiento del *EditText*, alguno de los valores más frecuentes:

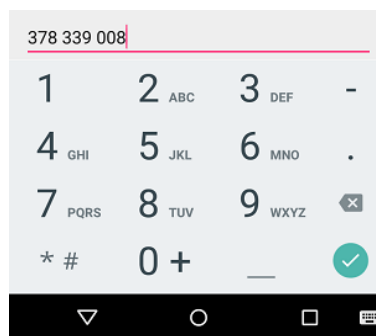
Constante	Descripción
<code>text</code>	Recibe texto plano simple
<code>textPersonName</code>	Texto correspondiente al nombre de una persona
<code>textPassword</code>	Protege los caracteres que se van escribiendo con puntos
<code>numberPassword</code>	Contraseña de solo números enmascarada con puntos
<code>textEmailAddress</code>	Texto que será usado en un campo para emails
<code>phone</code>	Texto asociado a un número de teléfono
<code>textPostalAddress</code>	Para ingresar textos asociados a una dirección postal
<code>textMultiLine</code>	Permite múltiples líneas en el campo de texto
<code>time</code>	Texto para determinar la hora
<code>date</code>	Texto para determinar la fecha
<code>number</code>	Texto con caracteres numéricos
<code>numberSigned</code>	Permite números con signo
<code>numberDecimal</code>	Para ingresar números decimales

Además de evitar que se escriban dichos caracteres, este atributo determina el tipo de teclado virtual que aparecerá ante el usuario y determina otros tipos de comportamientos.

Vamos a añadir otro campo que tenga como *hint* el valor Teléfono y que tenga como tipo de entrada de datos un teléfono:

```
} <EditText
    android:id="@+id/telefono"
    android:layout_width="match_parent"
    android:inputType="phone"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:hint="Teléfono"
    android:layout_below="@+id/campo_texto"/>
```

El resultado en ejecución será algo parecido a:



Para forzar el tamaño del texto que recibirá el *EditText* usa el atributo *android:maxLength*.

Especifica un número entero positivo para determinar cuántos caracteres podrá haber. Este atributo es de gran utilidad cuando las reglas de negocio indican restricciones a las entradas del usuario.

Añadiendo en la definición anterior *android:maxLength="12"*, el campo no permitirá ingresar más de 12 caracteres.

Es posible también el forzar que el *EditText* tenga una sola línea y no varias. Para ello hay que utilizar el atributo *android:singleLine*, por defecto el valor es *false* (más de una línea), por lo que hay que forzar su valor a *true* para conseguir este efecto.

Si deseas modificar el color del texto que especificas en *android:hint* usa el atributo *android:textColorHint*.

Vamos a modificar el color del *hint* del *EditText* asociado al teléfono

```
android:textColorHint="@color/colorPrimary"
```

Si deseas aumentar o reducir el tamaño del texto de un *EditText* implementa el atributo *android:textSize*.

Es posible también la personalización del color del borde inferior y el color del texto seleccionado dentro del control. Todo esto se hace a través de la definición de un estilo propio y asignando valores a dos propiedades *colorControlNormal* y *colorControlActivated*.

Abrimos el archivo *styles.xml* y definimos el siguiente estilo:


```
<style name="CampoTextoPurpura" parent="Theme.AppCompat.Light">
    <item name="colorControlNormal">#CE93D8</item>
    <item name="colorControlActivated">#AB47BC</item>
    <item name="android:textColorHighlight">#E1BEE7</item>
</style>
```

Asignamos el estilo al control que queramos, en nuestro caso al *EditText* del teléfono:

```
android:theme="@style/CampoTextoPurpura"
```

El foco es una característica de los views que determina si están activos hacia el usuario. Para los *EditTexts* dicho estado se representa cuando activan el cursor para la escritura y su borde inferior cambia el color. El atributo asociado a esta propiedad es *android:focusable*, cuyo valor por defecto es *true*. A nivel de programación se puede modificar el valor de este atributo con el método *setFocusable()*.

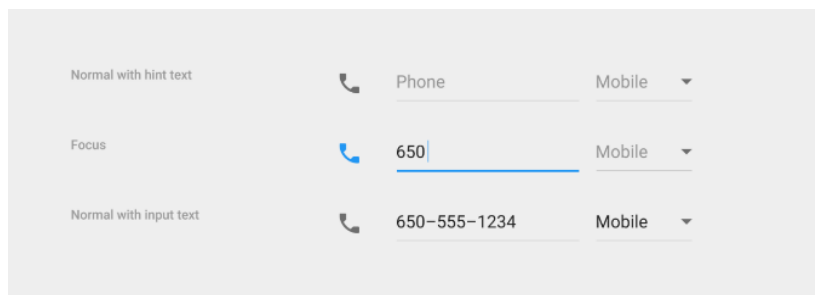
Si deseas darle el foco a un campo de texto programáticamente usa el método *requestFocus()* de la clase *View*.

Al igual que ocurría con los botones, donde podíamos indicar una imagen que acompañara al texto del mismo, con los controles de texto podemos hacer lo mismo. Las propiedades *drawableLeft* o *drawableRight* nos permite especificar una imagen, a izquierda o derecha, que permanecerá fija en el cuadro de texto.

Descargamos de los recursos de *Material Design* el icono *ic_call* y lo incluimos dentro del *EditText* del teléfono.

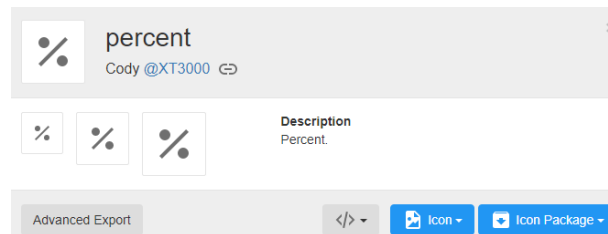
```
<EditText
    android:id="@+id/telefono"
    android:layout_width="match_parent"
    android:inputType="phone"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:hint="Teléfono"
    android:layout_below="@+id/campo_texto"
    android:maxLength="12"
    android:textColorHint="@color/colorPrimary"
    android:theme="@style/CampoTextoPurpura"
    android:drawableLeft="@drawable/ic_call_black"/>
```

La guía del *Material Design* para campos de texto determina que si vas a usar un icono para describir el contenido de un *EditText* con una sola línea, se debe cambiar el tinte de la imagen como reacción:



Para poder aplicar este efecto es recomendable la combinación de un *ImageView* y el *EditText*.

Vamos a descargar el icono siguiente con porcentaje de gris y tamaño 18dp (guardamos en *res/drawable*):



Creamos el archivo *activity_main_v2* y definimos el siguiente *layout*:

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:focusableInTouchMode="true"
    android:orientation="vertical"
    android:padding="@dimen/activity_horizontal_margin">

    <EditText
        android:id="@+id/campo_descuento"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_gravity="center_horizontal"
        android:layout_toEndOf="@+id/icono_descuento"
        android:layout_toRightOf="@+id/icono_descuento"
        android:inputType="numberDecimal"
        android:hint="Introduce descuento"/>

    <ImageView
        android:id="@+id/icono_descuento"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"
        android:layout_marginEnd="24dp"
        android:layout_marginRight="24dp"
        android:src="@drawable/ic_percent_grey" />
</RelativeLayout>
```

Cabe aclarar que usé el atributo *android:focusableInTouchMode* sobre el *RelativeLayout* para darle el foco en el momento que el usuario entre en *Touch Mode*. Esto evita que nuestro edit text sea el primero en tener el foco y no podamos ver la interacción de este ejemplo.

Definiremos ahora una escucha anónima para cuando nuestro control cambie el foco. Si tiene el foco actuaremos modificando el color del *EditText* y de la imagen asociada al mismo.

Estas acciones son fáciles si tienes claros los siguientes conceptos:

getDrawable(): Obtiene el objeto que representa el gráfico del icono.

DrawableCompat: Nueva clase de compatibilidad que permite cambiar cualidades de un *Drawable*. Usa el método *wrap()* para crear una copia del *drawable* y luego *setTint()* para cambiar el color del filtro.

ContextCompat: Clase de compatibilidad que te permitirá obtener recursos del proyecto. Usarás su método *getColor()* para extraer el recurso *R.color.colorAccent* definido en *colors.xml*.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_v2);
        EditText campoDescuento = (EditText) findViewById(R.id.campo_descuento);
        campoDescuento.setOnFocusChangeListener(new View.OnFocusChangeListener() {
            @Override
            public void onFocusChange(View v, boolean hasFocus) {
                ImageView iconoDescuento = (ImageView) findViewById(R.id.icono_descuento);
                Drawable d = iconoDescuento.getDrawable();
                d=DrawableCompat.wrap(d);
                if (hasFocus)
                    DrawableCompat.setTint(d, ContextCompat.getColor(getApplicationContext(), R.color.colorAccent));
                else
                    DrawableCompat.setTint(d, ContextCompat.getColor(getApplicationContext(), R.color.colorIconos));
            }
        });
    }
}
```

IMPORTANTE! En el caso de utilizar *setTint* para modificar el estado del *Drawable* no volverá al estado anterior por defecto, así que deberemos actualizarlo nosotros. Una técnica que podemos usar, es cambiar con *setTint* a su color inicial en el siguiente *onFocusChange*, para ello debemos saber cual es el color inicial de la imagen por lo que es recomendable usar imágenes en formato XML, donde en su atributo *fillColor* podemos ver su color exacto o incluso asignarle uno que tengamos definido en nuestros recursos de color (soportado con sdk mínimo 21).

```
<path android:fillColor="@color/colorIconos"
```

Imagen con parte del código xml del recurso de imagen percent.xml

En ocasiones necesitarás realizar tareas justo en el momento en que cambia el texto de un *EditText*. La interfaz que resuelve este tipo de situaciones se llama *TextWatcher* y se agrega a un campo de texto con el método *addTextChangedListener()*.

Esta clase te provee los siguientes controladores:

afterTextChanged(): Se llama cuando el cambio ya ha sido realizado. Esto permitirá acceder al texto que quedó luego del resultado.

beforeTextChanged(): Se llama antes de que se escriba el texto. Esto te permite saber el estado del texto actual y de la sección que será reemplazada.

onTextChanged(): Se llama cuando se ha reemplazado la sección del texto. Con sus parámetros permite saber qué porción del texto viejo se reemplazó y cuantos caracteres nuevos se agregaron.

Vamos a utilizar estos métodos para calcular en tiempo real la cantidad de caracteres de un campo de texto. Creamos otro archivo XML llamado *main_activity_v3* al que añadiremos un *EditText* (máximo de tres líneas) y *TextView* donde se visualizarán la cantidad de caracteres que vamos introduciendo en el *EditText*.

Con el atributo *android:inputType="textMultiLine"* declaras que el *EditText* tiene esta característica. Para delimitar visualmente tres, utilizamos el atributo *android:lines="3"* y con *android:maxLines="3"* limitamos el tamaño vertical.

Veamos la definición de este *layout*:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/activity_horizontal_margin">

    <EditText
        android:id="@+id/campo_mensaje"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:layout_gravity="center_horizontal"
        android:inputType="textMultiLine"
        android:lines="3"
        android:maxLines="3" />

    <TextView
        android:id="@+id/texto_contador"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignEnd="@+id/campo_mensaje"
        android:layout_alignRight="@+id/campo_mensaje"
        android:layout_below="@+id/campo_mensaje"
        android:text="Small Text"
        android:textAppearance="?android:attr/textAppearanceSmall" />
</RelativeLayout>
```

El objetivo es contar los caracteres del contenido del *EditText*, así que tanto *afterTextChanged()* u *onTextChanged()* pueden ser utilizados para esta tarea, ya que ambos proporcionan la cadena actual del campo de texto. Debido a que no es necesario saber el tamaño de la cadena actual o su posición inicial, nos decantaremos por *afterTextChanged()*.

```

EditText campoMensaje = (EditText) findViewById(R.id.campo_mensaje);
campoMensaje.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence charSequence, int i, int il, int i2) {

    }

    @Override
    public void onTextChanged(CharSequence charSequence, int i, int il, int i2) {

    }



    @Override
    public void afterTextChanged(Editable s) {
        TextView contador = (TextView) findViewById(R.id.texto_contador);
        String tamanoString = String.valueOf(s.length());
        contador.setText(tamanoString);
    }
});

```

Control de líneas aclaración

Android permite añadir una acción adicional al teclado para proporcionar una elección rápida cuando el usuario ha terminado de escribir su texto en el *EditText* (editor de método de entrada *IME*). Para indicar el tipo de acción usa el atributo *android:imeOptions*.

Alguna de las opciones más utilizadas son:

Constante	Descripción	Icono
<code>actionGo</code>	Representa la acción de ejecutar alguna tarea que llevará al usuario a determinados resultados	
<code>actionSearch</code>	Especifica que realizará una búsqueda con el texto que se acaba de agregar al campo de texto	
<code>actionSend</code>	Se usa para indicar que se realizará una operación de envío de un contenido asociado al contenido del campo	
<code>actionNext</code>	Tecla para realizar una operación del tipo "siguiente". Normalmente se usa para asignar el foco al <code>TextField</code> posterior	
<code>actionDone</code>	Determina que se ha llevado a cabo satisfactoriamente la edición cerrando el teclado virtual	
<code>actionPrevious</code>	Acción que lleva al usuario a un campo previamente aceptado.	

Para controlar los eventos e estos botones usaremos el escuchador *TextView.OnEditorActionListener* junto a su controlador *onEditorAction()*. Dentro de este

método agregaremos las instrucciones que deseamos ejecutar al presionar el botón de acción en el teclado.

Creamos el archivo XML *activity_principal_v4* y forzamos con el atributo *android:imeOptions="actionSearch"* a que aparezca en el teclado virtual esta opción.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/campo_busqueda"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:ems="10"
        android:hint="Buscar cliente"
        android:imeOptions="actionSearch"
        android:singleLine="true" />

</RelativeLayout>
```

Usaremos *TextView.OnEditorActionListener* junto a su controlador *onEditorAction()* para gestionar la pulsación de este botón. Mostramos un mensaje y cerramos teclado virtual.

```
EditText campoBusqueda = (EditText) findViewById(R.id.campo_busqueda);
campoBusqueda.setOnEditorActionListener(new TextView.OnEditorActionListener() {
    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
        boolean procesado = false;

        if (actionId == EditorInfo.IME_ACTION_SEARCH) {
            // Mostrar mensaje
            Toast.makeText(getApplicationContext(),
                "Acción búsqueda:" + v.getText().toString(), Toast.LENGTH_LONG).show();

            // Ocultar teclado virtual
            InputMethodManager imm = (InputMethodManager) getSystemService(INPUT_METHOD_SERVICE);
            imm.hideSoftInputFromWindow(v.getWindowToken(), 0);

            procesado = true;
        }
        return procesado;
    }
});
```

🔧 Ejercicio Resuelto HolaCualquiera

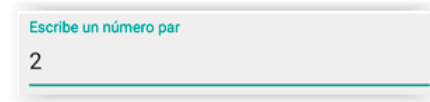
🔧 Ejercicio Propuesto HolaCualquiera segunda parte (implementando la interfaz *onClickListener* y un botón de despedida)

Etiquetas Flotantes (Floating Labels)

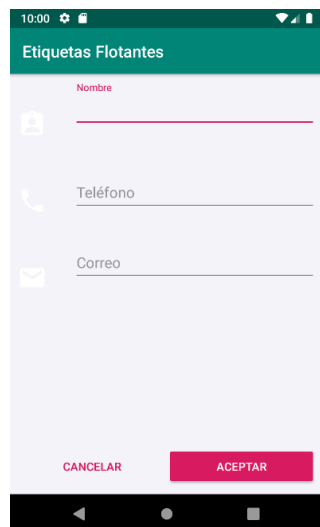
Dentro de la nueva librería de diseño (Design Support Library) tenemos este nuevo componente *TextInputLayout*, relacionado con los cuadros de texto que se mencionan en las especificaciones de *Material Design*. No es más que un *hint* que, en vez de desaparecer, se desplaza automáticamente a la parte superior del cuadro de texto cuando el usuario pulsa sobre él.

Para implementar *TextInputLayout* primero añadiremos la librería de diseño a nuestro proyecto y después habrá que añadir un contenedor de tipo *TextInputLayout* que incluirá en su interior la vista para insertar texto *TextInputEditText*.

```
<com.google.android.material.textfield.TextInputLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <com.google.android.material.textfield.TextInputEditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Escribe un número par"/>
</com.google.android.material.textfield.TextInputLayout>
```



Vamos a realizar un ejemplo más completo y ver sobre él algunas funcionalidades de este control. Los colores de la aplicación serán los que estén definidos por defecto.



Creemos un nuevo proyecto que llamaremos *proyectoEtiquetasFlotantes* y definimos el *layout* principal de la siguiente forma, definiremos dentro del contenedor principal (*RelativeLayout*) varios *LinearLayout*, uno para cada uno de los tres campos de entrada y otro para la zona de botones. No olvidar incluir la librería de *design*:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:padding="10dp"
    android:background="#199e"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout android:id="@+id/area_nombre"
        android:layout_weight="10"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:orientation="horizontal">

        <ImageView android:id="@+id/img_cliente"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:src="@drawable/ic_cliente" />

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/til_nombre"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginLeft="32dp">

            <com.google.android.material.textfield.TextInputEditText android:id="@+id/campo_nombre"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:ems="10"
                android:hint="@string/hint_nombre"
                android:inputType="text" />

        </com.google.android.material.textfield.TextInputLayout>
    </LinearLayout>

```

Añadir android:background="#199e"

Para diferenciar el color y que se vean mejor las imágenes

```

<LinearLayout android:id="@+id/area_telefono"
    android:layout_weight="10"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/area_nombre"
    android:orientation="horizontal">

    <ImageView android:id="@+id/img_correo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:src="@drawable/ic_telefono" />

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/til_telefono"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="32dp">

        <com.google.android.material.textfield.TextInputEditText android:id="@+id/campo_telefono"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:ems="10"
            android:hint="@string/hint_telefono"
            android:inputType="phone" />

    </com.google.android.material.textfield.TextInputLayout>
</LinearLayout>

```



```

<LinearLayout android:id="@+id/area_correo"
    android:layout_weight="10"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView android:id="@+id/img_telefono"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_column="0"
        android:layout_gravity="center_vertical"
        android:layout_row="2"
        android:src="@drawable/ic_correo" />

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/til_correo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="32dp">

        <com.google.android.material.textfield.TextInputEditText android:id="@+id/campo_correo"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:ems="10"
            android:hint="@string/hint_correo"
            android:inputType="textEmailAddress" />

    </com.google.android.material.textfield.TextInputLayout>
</LinearLayout>

```

```

<!-- Bottom Bar -->
<LinearLayout
    android:layout_weight="50"
    android:id="@+id/bottom_bar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:layout_alignParentBottom="true"
    android:gravity="bottom"
    android:orientation="horizontal">

    <Button
        android:id="@+id/boton_cancelar"
        style="@style/Widget.AppCompat.Button.Borderless.Colored"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/accion_cancelar" />

    <Button
        android:id="@+id/boton_aceptar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:backgroundTint="@color/colorAccent"
        android:text="@string/accion_aceptar"
        android:textColor="@android:color/white" />

</LinearLayout>
/LinearLayout>

```

Recordar crear los recursos en *strings* y descargar los *drawables*.

El control ofrece también la posibilidad de mostrar errores (muy útiles por ejemplo para mostrar errores de validación) bajo el cuadro de texto. Para ello podemos utilizar los métodos `setErrorEnabled(true)` y `setError()`. El primero de ellos reservará espacio debajo del cuadro de texto para mostrar los errores. El segundo nos servirá para indicar el texto del error o para eliminarlo (pasando null como parámetro).

El ejemplo mostrará los errores necesarios cuando se validen los campos de texto, al presionar el botón *GUARDAR*. Implementamos el *listener* para la acción *onClick()* del botón guardar:

Para validar el texto lo haremos a través de la clase Pattern, la cual contiene métodos para el uso de expresiones regulares.

```
public class MainActivity extends AppCompatActivity {

    private TextInputLayout tilNombre;
    private TextInputLayout tilTelefono;
    private TextInputLayout tilCorreo;
    private TextInputEditText cNombre;
    private TextInputEditText cTelefono;
    private TextInputEditText cCorreo;
    @Override
} protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    tilNombre = findViewById(R.id.til_nombre);
    tilTelefono = findViewById(R.id.til_telefono);
    tilCorreo = findViewById(R.id.til_correo);
    cNombre= findViewById(R.id.campo_nombre);
    cTelefono=findViewById(R.id.campo_telefono);
    cCorreo=findViewById(R.id.campo_correo);
    Button botonAceptar = (Button) findViewById(R.id.boton_aceptar);
} botonAceptar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        validarDatos();
    }
});
Button botonCancelar=(Button) findViewById(R.id.boton_cancelar);
}
} private boolean esNombreValido(String nombre) {
    Pattern patron = Pattern.compile("[a-zA-Z ]+$");
    if (!patron.matcher(nombre).matches() || nombre.length() > 30) {
        tilNombre.setError("Nombre inválido");
        return false;
    } else {
        tilNombre.setError(null);
    }
    return true;
}
}
```

```

    }
    private boolean esTelefonoValido(String telefono) {
        if (!Patterns.PHONE.matcher(telefono).matches()) {
            tilTelefono.setError("Teléfono inválido");
            return false;
        } else {
            tilTelefono.setError(null);
        }

        return true;
    }

    private boolean esCorreoValido(String correo) {
        if (!Patterns.EMAIL_ADDRESS.matcher(correo).matches()) {
            tilCorreo.setError("Correo electrónico inválido");
            return false;
        } else {
            tilCorreo.setError(null);
        }

        return true;
    }

    private void validarDatos() {
        String nombre = tilNombre.getText().getText().toString();
        String telefono = tilTelefono.getText().getText().toString();
        String correo = tilCorreo.getText().getText().toString();

        boolean a = esNombreValido(nombre);
        boolean b = esTelefonoValido(telefono);
        boolean c = esCorreoValido(correo);

        if (a && b && c) {
            // OK, se pasa a la siguiente acción
            Toast.makeText(this, "Se guarda el registro", Toast.LENGTH_LONG).show();
        }
    }
}

```

iii Implementar el botón cancelar

Etiquetas predictivas: (AutoCompleteTextView)

Este control permite que dentro del mismo aparezcan sugerencias que faciliten la introducción de datos. Las sugerencias que propone la vista a medida que se escribe se suministran en el código java de la *activity* que contiene la vista. Estas están contenidas en un array que se pasa como argumento a un adaptador. Crear proyecto *proyectoAutoCompleteText*.

¿Pero qué es un adaptador? Un adaptador es un intermediario entre los datos a mostrar y la vista que los va a representar. El primer argumento del adaptador es el contexto, nuestra *activity* en este caso. El segundo es el identificador del recurso del fichero de diseño en el que mostrar la vista. En este caso es un diseño predefinido asociado aun *TextView*. El tercer parámetro son los datos a mostrar.

```

package com.example.director.ejemploautocompletetextview;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;

public class MainActivity extends AppCompatActivity {

    private static final String[] CIUDADES = new String[] {
        "Burgos", "Soria", "Barcelona", "Sevilla", "Santander", "Alicante", "Almería"
    };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
            android.R.layout.simple_dropdown_item_1line, CIUDADES);

        AutoCompleteTextView textView = (AutoCompleteTextView) findViewById(R.id.autoCompleteTextView1);
        textView.setAdapter(adapter);
    }
}

```

Finalmente se asigna el adaptador a la vista, en este caso a la variable *textView*. La vista podría ser como la que sigue, el atributo *completionThreshold* indica a partir de cuantos caracteres introducidos, comienza la búsqueda.

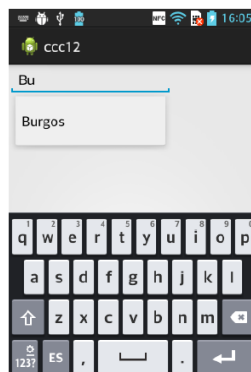
```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="com.example.profesor.proyectoautocompletetext.MainActivity">

    <AutoCompleteTextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Ciudad"
        android:completionThreshold="2"/>

</RelativeLayout>

```



Añadir el atributo *android:ems="10"* al control para qe la visualización sea la correcta.

Nota ampliativa

Una alternativa a tener en cuenta si los datos a mostrar en el control son estáticos sería definir la lista de posibles valores como un recurso de tipo *string-array*. Para ello, primero crearíamos un nuevo fichero XML en la *carpeta/res/values* llamado por ejemplo *ciudades.xml* e incluiríamos en él los valores seleccionables de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="nombre_ciudades">
        <item>Burgos</item>
        <item>Soria</item>
        <item>Barcelona</item>
        <item>Sevilla</item>
        <item>Santander</item>
        <item>Alicante</item>
        <item>Almería</item>
    </string-array>
</resources>
```

Tras esto, a la hora de crear el adaptador, utilizaríamos el método *createFromResource()* para hacer referencia a este array XML que acabamos de crear:

```
package com.example.profesor.proyectoautocompletetext;

import ...

public class MainActivity extends AppCompatActivity {
    private static final String[] CIUDADES=new String[] {"Burgos", "Soria", "Barcelona", "Sevilla",
        "Santander", "Alicante", "Almería"};
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /*ArrayAdapter<String> adapter=new ArrayAdapter<String>(this,
            android.R.layout.simple_dropdown_item_1line,CIUDADES);*/

        ArrayAdapter<CharSequence> adapter=ArrayAdapter.createFromResource(this,
            R.array.nombre_ciudades,android.R.layout.simple_spinner_item);
        AutoCompleteTextView textView=(AutoCompleteTextView) findViewById(R.id.texto_ciudades);
        textView.setAdapter(adapter);
    }
}
```

CHECKBOX.

Un Checkbox es un botón de dos estados (marcado, no marcado) que actúa como control de selección y que permite elegir una o varias opciones de un conjunto.

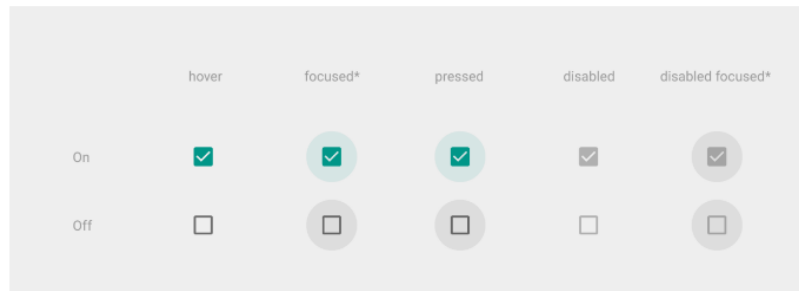
Por defecto su color será el mismo de la propiedad `android:colorAccent` en el tema de la aplicación.

Las especificaciones de MaterialDesign nos muestra que este view puede pasar por 5 estados, tanto si está marcado como si no:

- **hover**: Estado cuando el Checkbox se encuentra inmóvil
- **focused**: Cuando la navegación en la UI apunta al checkbox como control de entrada
- **pressed**: El checkbox se encuentra bajo selección prolongada
- **disable**: El checkbox pierde su capacidad de cambio entre estados

- **disable-focused:** El checkbox está deshabilitado pero el sistema de navegación lo enfoca

Las siguientes ilustraciones muestran los estados descritos anteriormente en el tema Material Light:



En Android está representado por la clase del mismo nombre, `CheckBox`. La forma de definirlo en nuestra interfaz y los métodos disponibles para manipularlos desde nuestro código son análogos a los ya comentados para el control `ToggleButton`.

Para explicar este control, crear un nuevo proyecto con el nombre `CheckBox` y definir la actividad principal como sigue:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <CheckBox
        android:id="@+id/opcion_mostrar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/campo_contrasena"
        android:layout_alignStart="@+id/campo_contrasena"
        android:layout_below="@+id/campo_contrasena"
        android:onClick="mostrarContraseña"
        android:text="¿Mostrar contraseña?" />

    <EditText
        android:id="@+id/campo_contrasena"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:ems="10"
        android:hint="Contraseña"
        android:inputType="textPassword" />
</RelativeLayout>
```

La vista que obtendremos será:



Para manejar eventos sobre esta *View* procederemos de alguna de las formas vistas anteriormente, o bien asignando el evento en la propiedad *onClick* del archivo XML o bien con un escuchador anónimo.

El ejemplo que vamos a desarrollar es la visualización o no de la contraseña introducida en un *TextView* dependiendo del estado de selección de un *CheckBox*.

```
package com.example.profesor.checkbox;

import ...

public class MainActivity extends AppCompatActivity {

    private CheckBox opcionMostrar;
    private EditText campoContrasena;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        opcionMostrar = (CheckBox)findViewById(R.id.opcion_mostrar);
        campoContrasena = (EditText)findViewById(R.id.campo_contrasena);
        /*opcionMostrar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mostrarContraseña(v);
            }
        });*/

    }

    public void mostrarContraseña(View v){
        // Salvar cursor
        int cursor = campoContrasena.getSelectionEnd();

        if(opcionMostrar.isChecked()){
            campoContrasena.setInputType(InputType.TYPE_CLASS_TEXT
                | InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD);
        }else{
            campoContrasena.setInputType(InputType.TYPE_CLASS_TEXT
                | InputType.TYPE_TEXT_VARIATION_PASSWORD);
        }
        // Restaurar cursor
        campoContrasena.setSelection(cursor);
    }
}
```

Para hacer visible o no la contraseña lo único que debemos tener en cuenta es el valor de *InputType* del control *TextView* correspondiente, asignándole uno de los dos valores que

aparecen en el código según el estado del *CheckBox*. Para detectar el valor del *CheckBox* lo haremos con el método *isChecked()*.

Otro posible evento que puede lanzar este control es sin duda el que informa de que ha cambiado el estado del control, que recibe el nombre de ***onCheckedChanged***. Para implementar las acciones de este evento podríamos utilizar por tanto la siguiente lógica:

```
opcionMostrar.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {  
    @Override  
    public void onCheckedChanged(CompoundButton compoundButton, boolean b) {  
        if(opcionMostrar.isChecked()){  
            Toast.makeText(getApplicationContext(),"Control chequeo TRUE",Toast.LENGTH_LONG).show();  
        }else{  
            Toast.makeText(getApplicationContext(),"Control chequeo FALSE",Toast.LENGTH_LONG).show();  
        }  
    }  
});
```

RADIOBUTTON

Un *RadioButton* es un control de selección que proporciona la interfaz de Android para permitir a los usuarios seleccionar una opción de un conjunto.

Son ideales para elegir uno de varios elementos con exclusión mutua, es decir, la selección de un radio button obliga a descartar la de otro, permitiendo solo a un ítem estar activo.

En Android, un grupo de botones *radiobutton* se define mediante un elemento *RadioGroup*, que a su vez contendrá todos los elementos *RadioButton* necesarios. Veamos un ejemplo de cómo definir un grupo de dos controles *radiobutton* en nuestra interfaz:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical"  
    android:padding="@dimen/activity_horizontal_margin">  
    <RadioGroup  
        android:id="@+id/rg_tipo_cliente"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_marginBottom="24dp"  
        android:checkedButton="@+id/rb_particular"  
        android:orientation="vertical">  
        <RadioButton  
            android:id="@+id/rb_corporativo"  
            android:layout_width="wrap_content"  
            android:layout_height="wrap_content"  
            android:text="Corporativo" />  
        <RadioButton  
            android:id="@+id/rb_particular"  
            android:layout_width="wrap_content"  
            android:layout_height="wrap_content"  
            android:text="Particular" />  
    </RadioGroup>  
</LinearLayout>
```

En primer lugar vemos cómo podemos definir el grupo de controles indicando su orientación (vertical u horizontal) al igual que ocurriría por ejemplo con un *LinearLayout*. Tras esto, se

añaden todos los objetos `RadioButton` necesarios indicando su ID mediante la propiedad `android:id` y su texto mediante `android:text`.

Una vez definida la interfaz podremos manipular el control desde nuestro código java haciendo uso de los diferentes métodos del control `RadioGroup`, los más importantes: **`check(id)`** para marcar una opción determinada mediante su ID, **`clearCheck()`** para desmarcar todas las opciones, y **`getCheckedRadioButtonId()`** que como su nombre indica devolverá el ID de la opción marcada (o el valor -1 si no hay ninguna marcada).

En cuanto a los eventos lanzados, al igual que en el caso de los checkboxes, los más importantes serán el que informa de los cambios en el elemento seleccionado, llamado también en este caso **`onCheckedChangeListener`** y el escuchador correspondiente para detectar el evento Click sobre la vista.

Vemos cómo tratar el primer evento del objeto `RadioGroup`:

```
package com.example.profesor.radiobutton;

import ...

public class MainActivity extends AppCompatActivity {
    private RadioGroup rg;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        rg=(RadioGroup)findViewById(R.id.rg_tipo_cliente);

        rg.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup radioGroup, int i) {
                Toast.makeText(getApplicationContext(), "Opción chequeada:"|i, Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

En la documentación de `MaterialDesign` especifican que un `Radio Button` puede atravesar por 5 estados al igual que los otros controles de selección.

- **hover**: Estado cuando el `Checkbox` se encuentra inmóvil
- **focused**: Cuando la navegación en la UI apunta al `checkbox` como control de entrada
- **pressed**: El `checkbox` se encuentra bajo selección prolongada
- **disable**: El `checkbox` pierde su capacidad de cambio entre estados
- **disable-focused**: El `checkbox` está deshabilitado pero el sistema de navegación lo enfoca

Realizar el EjercicioResueltoRadioButton

EjercicioPropuesto

TOAST

Los *Toast* sirven para mostrar al usuario algún tipo de información de la manera menos intrusiva posible, sin robar el foco a la actividad en ejecución y sin pedir ningún tipo de interacción con el usuario, desapareciendo automáticamente. El tiempo que permanecerá en pantalla puede ser, o bien *Toast.LENGTH_SHORT*, o bien *Toast.LENGTH_LONG*. Se crean y muestran as

```
Toast.makeText(MiActividad.this,"Este es el mensaje", Toast.LENGTH_SHORT).show();
```

No hay que abusar de ellos pues, si se acumulan varios, irán apareciendo uno después de otro, esperando a que se acabe el anterior y quizás a destiempo. Son útiles para conformar algún tipo de información al usuario, dándole la seguridad de que está haciendo lo correcto.

También podemos personalizarlo un poco cambiando su posición en la pantalla. Para esto utilizaremos el método *setGravity()*, al que podremos indicar en qué zona deseamos que aparezca la notificación. La zona deberemos indicarla con alguna de las constantes definidas en la clase *Gravity*: *CENTER*, *LEFT*, *BOTTOM*, ... o con alguna combinación de éstas.

```
toast2.setGravity(Gravity.CENTER|Gravity.LEFT,0,0);
```

Es posible personalizar la apariencia de la notificación *toast*, esto se haría a través de la definición de un *layout* XML específico para el *toast*, donde se pueden incluir todos los elementos necesarios para adaptar la notificación a nuestras necesidades. Veamos cómo hacerlo. Primero definimos el *layout* personalizado, por ejemplo el siguiente.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/lytLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <ImageView
        android:layout_width="50dp"
        android:layout_height="25dp"
        android:src="@drawable/marcador"
        android:id="@+id/imgIcono"/>

    <TextView
        android:id="@+id/txtMensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:paddingLeft="10dp"/>
</LinearLayout>
```

Luego introducimos el código java.

```

package com.example.profesor.toastpersonalizado;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button botonT=(Button) findViewById(R.id.botonToast);
        botonT.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast toast=new Toast(getApplicationContext());

                LayoutInflater inflater=getLayoutInflater();
                View layout=inflater.inflate(R.layout.toast_layout, (ViewGroup)findViewById(R.id.lytLayout));

                TextView txtMsg=(TextView) layout.findViewById(R.id.txtMensaje);
                txtMsg.setText("Texto que aparece en el Toast");

                toast.setDuration(Toast.LENGTH_LONG);
                toast.setView(layout);
                toast.show();
            }
        });
    }
}

```

En primer lugar debemos “inflar” nuestro *layout* mediante el objeto *LayoutInflater*, como vemos en la línea 20 y 21, donde pasamos como parámetro en nombre del recurso XML que contiene nuestro *layout* para el *toast* (*R.layout.toast_layout*) y el nombre de dicho *layout* dentro de la aplicación (*R.id.lytLayout*).

SNACKBAR

Es un nuevo tipo de notificación, que ha tomado especial relevancia sobre todo a raíz de la aparición de Android 5 Lollipop y Material Design, son los llamados *snackbar*. Un *snackbar* debe utilizarse para mostrar feedback sobre alguna operación realizada por el usuario, y es similar a un *toast* en el sentido de que aparece en pantalla por un corto periodo de tiempo y después desaparece automáticamente, aunque también presenta algunas diferencias importantes, como por ejemplo que puede contener un botón de texto de acción.

Un *snackbar* debe mostrar mensajes cortos, debe aparecer desde la parte inferior de la pantalla (con la misma elevación que el *floating action button* si existiera, pero menos que los diálogos o el *navigation drawer*), no debe mostrarse más de uno al mismo tiempo, puede contener un botón de texto para realizar una acción, y normalmente puede descartarse deslizándolo hacia un lateral de la pantalla.

Salvo el último punto, que aclararemos más adelante, el nuevo componente *Snackbar* de la librería de diseño se encargará de asegurar todos los demás, y además mantendrá la facilidad de uso de los *toast*, siendo su API muy similar. Para poder usarla deberemos añadir la referencia a la nueva librería de diseño a nuestro proyecto, como ya hemos hecho con otros componentes explicados en este tema.

Snackbar sencillo

Para mostrar un snackbar construiremos el objeto mediante el método estático `Snackbar.make()` y posteriormente lo mostraremos llamando al método `show()`, igual que en los Toast.

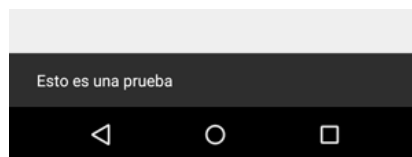
```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Snackbar sncBar=Snackbar.make(findViewById(R.id.activity_main),"Esto es una prueba",
                                     Snackbar.LENGTH_INDEFINITE);
        sncBar.show();
    }
}
```

El segundo y el tercer argumento son equivalentes a los ya mostrados para los toast, aunque la duración también puede ser `Snackbar.LENGTH_SHORT`, `Snackbar.LENGTH_LONG` o indefinida como se ve en la imagen.

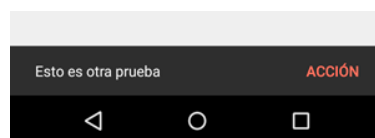
El primero argumento debe pasar la referencia a una vista que permita al snackbar descubrir un “contenedor adecuado” donde alojarse. Este contenedor será normalmente el content view o vista raíz de la actividad, o un contenedor de tipo `CoordinatorLayout`.



Snackbar con acción

Para añadir una acción al snackbar utilizaremos su método `setAction()`, al que pasaremos como parámetros el texto del botón de acción, y un listener para su evento `onClick` (análogo al de un botón normal) donde podremos responder a la pulsación del botón realizando las acciones.

```
Snackbar sncBar=Snackbar.make(findViewById(R.id.activity_main),"Esto es una prueba",
                             Snackbar.LENGTH_INDEFINITE);
sncBar.setActionTextColor(Color.rgb(22,99,00));
sncBar.setAction("Acción", new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Toast.makeText(getApplicationContext(),"Pulsaste la acción de SNACKBAR",
                        Toast.LENGTH_LONG).show();
    }
});
sncBar.show();
```



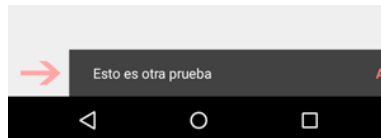
Una vez se pulse el botón de acción el `SnackBar` desaparecerá automáticamente. También hemos configurado el color con el que aparecerá el botón de acción.

Descartar Snackbar

Para descartar un snackbar con el gesto de deslizamiento hacia la derecha necesitaremos un contenedor llamado CoordinatorLayout. Este componente se utiliza para gestionar de una forma semiautomática algunas de las animaciones habituales de una aplicación android, y lo que es más importante, teniendo presente cómo la animación de algún elemento puede afectar a otros. Si añadimos como elemento padre de nuestro layout un CoordinatorLayout, nuestros snackbar podrán automáticamente descartarse deslizándolos a la derecha. Nuestro layout podría quedar por ejemplo de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/coordinator"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
    />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Si volvemos a ejecutar ahora la aplicación, veremos que todo sigue funcionando de la misma forma que antes, con el añadido de que podremos descartar los mensajes deslizándolos antes de que desaparezcan.



EjercicioPropuestoAgenda

ⁱ Este código permite controlar el número de líneas utilizadas en el control

```
final EditText campoMensaje = (EditText) findViewById(R.id.campo_mensaje);
campoMensaje.addTextChangedListener(new TextWatcher() {
    int lastSpecialRequestsCursorPosition=0;
    String specialRequests = "";
    @Override
    public void beforeTextChanged(CharSequence charSequence, int i, int i1, int i2) {
        lastSpecialRequestsCursorPosition = campoMensaje.getSelectionStart();
    }

    @Override
    public void onTextChanged(CharSequence charSequence, int i, int i1, int i2) {

    }

    @Override
    public void afterTextChanged(Editable s) {
        TextView contador = (TextView) findViewById(R.id.texto_contador);
        String tamanoString = String.valueOf(s.length());
        contador.setText(tamanoString);

        campoMensaje.removeTextChangedListener(this);
        if (campoMensaje.getLineCount() > campoMensaje.getMaxLines()) {
            campoMensaje.setText(specialRequests);
            campoMensaje.setSelection(lastSpecialRequestsCursorPosition);
        }
        else
            specialRequests = campoMensaje.getText().toString();

        campoMensaje.addTextChangedListener(this);
    }
});
```

API mínima 16 para referenciar bien los atributos del layout.

ⁱⁱ Implementación del botón cancelar:

```
Button botonCancelar=(Button) findViewById(R.id.boton_cancelar);
botonCancelar.setOnClickListener((view) → {
    Toast.makeText(getApplicationContext(), "Edición Cancelada", Toast.LENGTH_LONG).show();
    tilNombre.setError("");
    tilTelefono.setError("");
    tilCorreo.setError("");
    cNombre.setText("");
    cTelefono.setText("");
    cCorreo.setText("");
});
```